ENGR 418 PROJECT REPORT **School of Engineering**
Faculty of Applied Science
University of British Columbia

Project Title: Sorting Lego Pieces Using Feature Engineering Techniques and Logistic Regression
Group No.: 12
Members: Drayton Monkman, Spencer Szabo
Date: December 14, 2021

Introduction

The purpose of this project is to develop a machine learning algorithm using python to train and test a multiclass classifier. Three classes of images are provided in two separate datasets: "training" and "testing", but the images are not necessarily centered and are often at varying angles. Our objectives are to extract or engineer features from the different image classes and use relevant machine learning techniques and python modules to properly train weights based on the "training" data set. Using these weights, we are to accurately classify the different types of images from the "testing" dataset, and present our results in the form of both a confusion matrix and accuracy indicator. In this report, we implement the use of a canny edge detector and a threshold binary filter to find the edges of each image .We then perform principal component analysis (PCA) on the edge points of our converted images to find the eigenvalues corresponding to largest variance, and a logistic regression algorithm using our modified data to classify each image.

Theory

The classifier begins with advanced image processing. First, we algorithmically crop the image as much as safely possible to reduce noise. This is done using a canny filter to detect edge pixels, then calculating the centroid of those pixels, then cropping. When the cropping is complete, we move on to collect two filtered images that will be used to extract data.

The first is a canny filtered image of the cropped image. This image is converted to a 2D dataset and then mean centered so that we can perform principal component analysis. PCA generates a base for representing data with vectors pointing in the dataset's largest directions of orthonormal variance [1]. These vectors are the 'principal components' with the first principal component pointing in the direction of largest variance, and the second principal component pointing in the direction of largest variance such that it is orthogonal to the first [1]. These principal components are found by constructing a covariance matrix of data, and using that matrix to calculate the eigenvalue decomposition [1]. The difference in length of these vectors will tell us if a piece has a long (ie. rectangular) shape, or a more evenly balanced shape.

The second image is obtained by passing our cropped image through a mean threshold binary filter. We then take a contour from this image, and obtain the data from the largest continuous contour. This contour is a set of vertices describing the outline of the shape. These were then broken down into vectors that represent straight edges between points, and angles between each vector were calculated, summed and stored. This stored value of angles is then used in weighing our model. Shapes with long straight edges, like squares and rectangles, will see a disproportionate count of zero angles. Rounder shapes will see a more dispersed count of angles, and fewer counted zero angles.

We then pass the summed angle weights and eigenvector weights from the PCA into a logistic regression algorithm similar to what we used in our stage 1 report. Accurate training and testing can then be done to classify the images based on this extracted data.

Algorithm

There are two algorithms utilized for this project, the training algorithm and the testing algorithm. Although similar, they have subtle differences. First of all, we have the training algorithm, implemented in the "TrainShapeDetectionModel" function, which works as described by the following pseudocode:

```
1) Load a list of training data from the 'Lego_dataset_2/training' directory
2) Iterate through each image to perform the following operations:
    a. Load the training image
    b. Crop the image using cropping algorithm
    c. Extracting principal components from a canny (binary edge) filtered image
    d. Extracting edge data using threshold filtered image
    e. Determine true class based on filenames (for training purposes only)
3) Train a logistic regression model using sklearn.linear_model.LogisticRegression
4) Return the trained model
```

After completion of model training, our training function will then test the model against the training data and print performance metrics, namely a confusion matrix and an accuracy score. Presumably this will be a perfect or near perfect result.

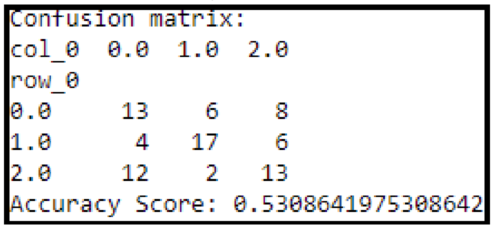
Secondly we have the testing algorithm, implemented in the "test_function" function, which works as described by the following pseudocode:

```
1) Iterating through each image to perform the following operations:
    a. Load the testing image
    b. Crop the image using cropping algorithm
    c. Extracting principal components from a canny (binary edge) filtered image
    d. Extracting edge data using threshold filtered image
    e. Determining true class based on filenames (optional, for testing accuracy only)
    f. Test against model
2) Compile resultant data for creation of classification report
```

The results of this algorithm are then displayed using a classification report and an accuracy score. With a well trained model, the accuracy score should be high, though likely not as high as when testing against the training data. Please note that both models are currently quite slow, and can take up to ten seconds per image process when running in a jupyter environment.

Results and Discussion

The training and testing image datasets for stage 2 were first run using our algorithm from stage 1. The resulting confusion matrix and accuracy score from the testing can be seen below.

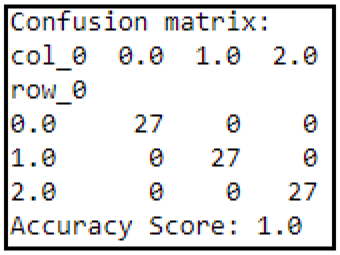


```
Confusion matrix:
col_0  0.0  1.0  2.0
row_0
0.0     13    6    8
1.0      4   17    6
2.0     12    2   13
Accuracy Score: 0.5308641975308642
```

Figure 1: Confusion Matrix and Accuracy Score of Stage 1 Model

The performance of our stage 1 algorithm on these new datasets was poor, with an accuracy score of 53%. This is due to our algorithm taking advantage of the fact that the images in the stage 1 datasets were generally centered and aligned consistently.

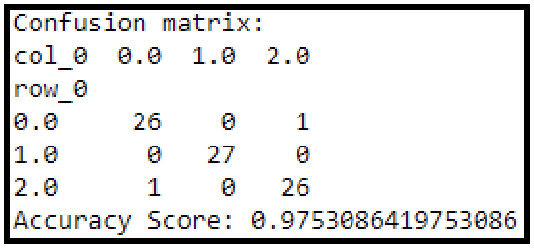
The performance of our stage 2 model drastically improved accuracy results.



```
Confusion matrix:
col_0  0.0  1.0  2.0
row_0
0.0     27    0    0
1.0      0   27    0
2.0      0    0   27
Accuracy Score: 1.0
```

Figure 2: Confusion Matrix and Accuracy Score of Model vs Training Data

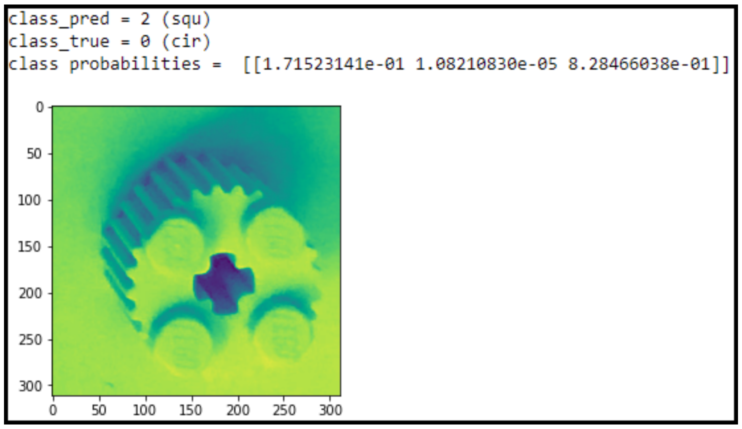
As shown in Figure 2, all images were correctly classified by our model resulting in a perfect accuracy score of 1.0. However, we cannot reasonably determine if our model is overfit when testing against the training data. To determine the true predictive accuracy of our model, we need to test against data not yet seen by the model.



```
Confusion matrix:
col_0  0.0  1.0  2.0
row_0
0.0     26    0    1
1.0      0   27    0
2.0      1    0   26
Accuracy Score: 0.9753086419753086
```

Figure 3: Confusion Matrix and Accuracy Score of Model vs Testing Data

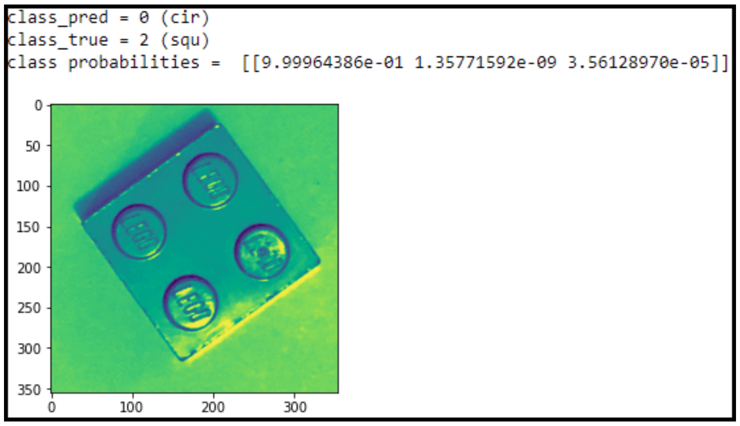
As shown in Figure 3, our model performed well against the testing data receiving an accuracy score of 0.975. Out of 81 test images, only 2 were misclassified.
One circle (class 0) was misclassified as a square (class 2), and one square (class 2) was misclassified as a circle (class 0). We can take a closer look at these cases and attempt to infer why they were misclassified.



```
class_pred = 2 (squ)
class_true = 0 (cir)
class probabilities =  [[1.71523141e-01 1.08210830e-05 8.28466038e-01]]
```

Figure 4: Misclassification 1 of Model vs Testing Data

In Figure 4 we see a circle (class 0) being misclassified as a square (class 2) with a confidence of 0.828. It is not good for a model to be confidently wrong as it is in this case. When analyzing the image closer, we can see that the picture's bottom edge is blending in with the background. This is problematic for our algorithm as it implements the use of edge detection to properly classify the photos. This is likely the main cause of this misclassification.



```
class_pred = 0 (cir)
class_true = 2 (squ)
class probabilities =  [[9.99964386e-01 1.35771592e-09 3.56128970e-05]]
```

Figure 5: Misclassification 2 of Model vs Testing Data

In Figure 5 we see a square (class 2) being misclassified as a circle (class 0) with a confidence that rounds up to 1.000. This is highly problematic, as the model is almost certain that the image should be a circle. After examining the result in detail, the bright reflection at the bottom of the piece causes the obtained contour to wrap into the middle of the piece and run along the edges of the round pegs. These rounded edges were likely the cause of misclassification.

Conclusions

In summary, we created a model that was capable of successfully predicting imperfectly placed lego shapes in testing images with an accuracy of 0.975. The used methods were overall successful, but could still be improved. The speed of the classifier was quite slow taking a few seconds per classification. There were likely more efficient ways we could have implemented our data extraction algorithm to increase processing speed. We also could have opted to extract other forms of information from the images. Despite these drawbacks and ways we could have improved, we are overall happy with how our model performed given the imperfect conditions.

Appendices

Appendix A: Software Used
Microsoft Windows 10, Jupyterlab, Python 3.8.5

Appendix B: Proprietary Python Packages Used
Scikit Learn, Scikit Image, pandas, matplotlib, NumPy

References

1. J. Watt, R. Borhani, and A. K. Katsaggelos, *Machine Learning Refined: Foundations, Algorithms, and Applications*. Cambridge: Cambridge University Press, 2016.